**Homework 3**

**Part 1**: netcat TCP chat

**We will begin with netcat—a TCP/UDP utility which comes preinstalled on most UNIX**

**systems. As quoted from the man page (open a terminal and type man netcat), "The nc (or**

**netcat) utility is used for just about anything under the sun involving TCP or UDP. It can open**

**TCP connections, send UDP packets, listen on arbitrary TCP and UDP ports, do port scanning,**

**and deal with both IPv4 and IPv6."**

**We will use netcat to create a two-way chat over a network. The purpose of this section is to**

**see an example of how easy it can be to implement a TCP chat over a network.**

**Let R2 be the server and KALI the client. Type the following commands:**

**R2 (Server):**

**nc -l <port> // use any port that is not reserved (e.g., 5000)**

**KALI (Client):**

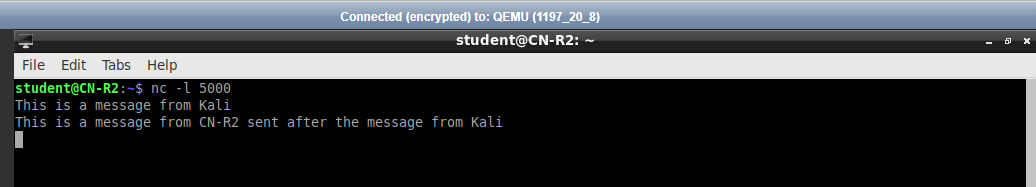
**nc <server hostname or IP address> <port>**

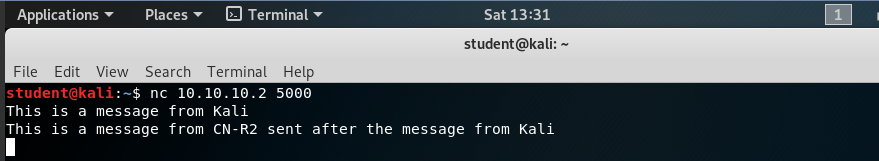
**On R2, type a message in the terminal window and press enter. It should be viewable in R3’s**

**terminal. Type a message in R3’s terminal window and press enter. It should be viewable in**

**R2’s terminal. Type a few messages on each side to make a conversation.**

**Press CTRL-C in both R2 and KALI to close the connection.**





The only problem with this chat is that it is hard to tell who sent each message. Our goal is for

the chat window to behave like this:

R2 types: “Hi, my name is R2”

Output in R2 and KALI terminal: “R2: Hi, my name is R2”

KALI types: “Hi R2, my name is KALI. Nice to meet you.”

Output in R2 and KALI terminal: “KALI: Hi R2, my name is KALI. Nice to meet

you.

Add a username (“R2” or “KALI”) so that each message can be identified by its sender.

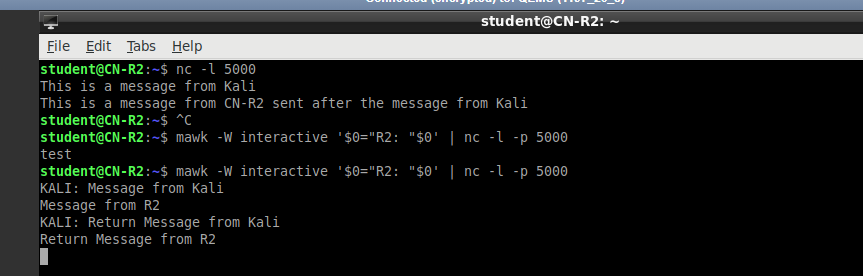
R2 (Server):

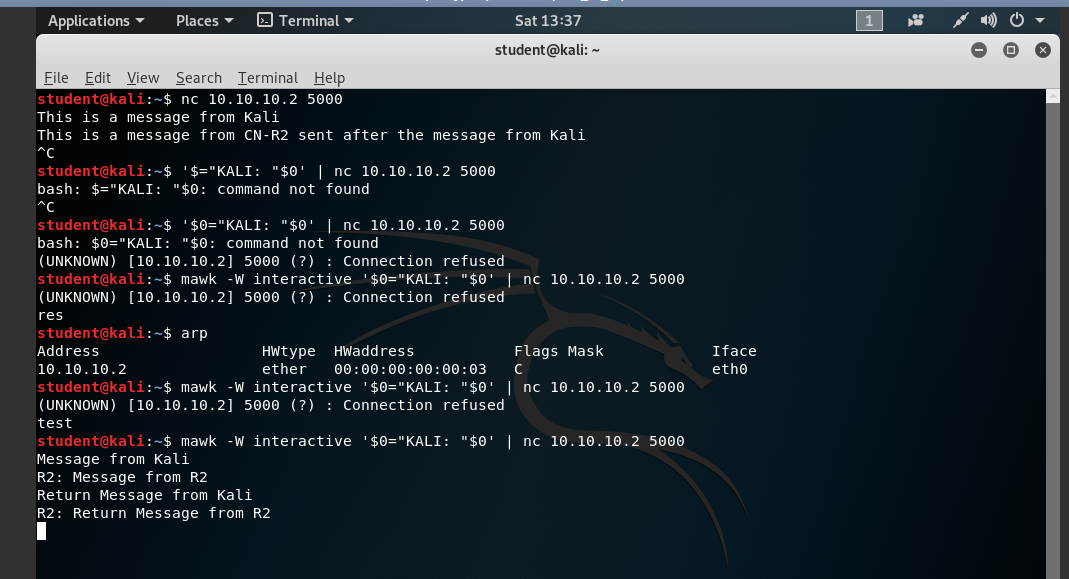
mawk -W interactive '$0="R2: "$0' | nc -l -p <port\_number>

KALI (Client):

mawk -W interactive '$0="KALI: "$0' | nc <server IP> <port\_number>

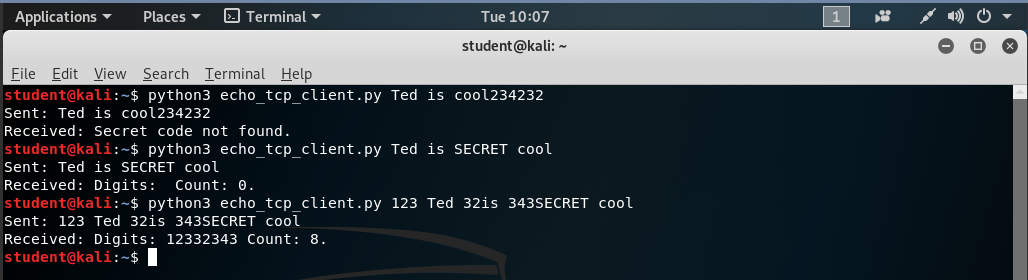
Create a short conversation between R2 and KALI and take a screenshot of each terminal

window showing the chat.



**Part 2:** **Client-server with secret code**

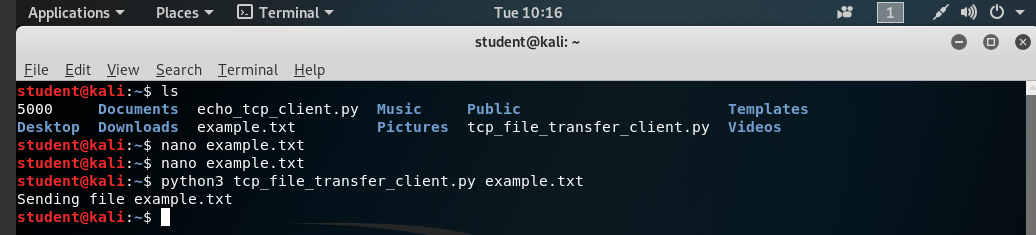
You should write two files for this part: **echo\_tcp\_server.py** (on R2) and **echo\_tcp\_client.py** (on KALI). The client should send a string to the server, and the server should receive it. If the string contains the secret code “SECRET”, the server should return all the digits in the string as well as the total number of digits. If the string does not contain the secret code, the server should respond with the message, “Secret code not found.” The client should receive the output. The output format and behavior should match the example below.



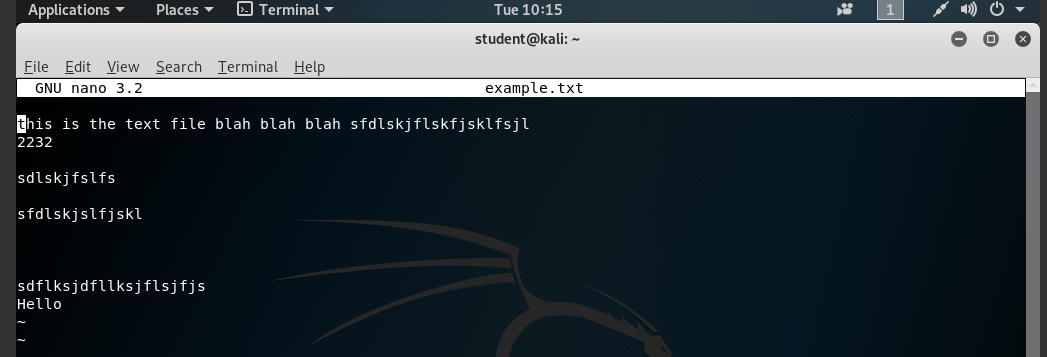


**Part 3: Client-server with file transfer**

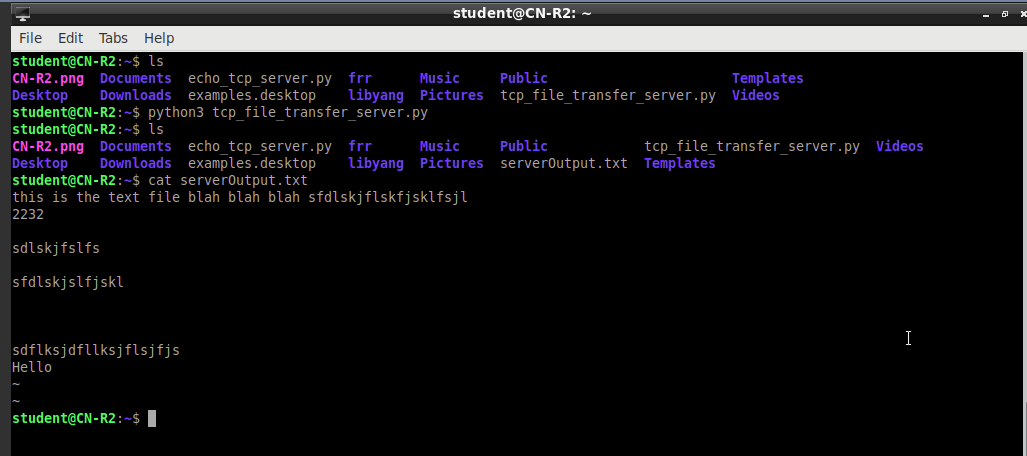
You should write two files for this part: **tcp\_file\_transfer\_server.py** (on R2) and **tcp\_file\_transfer\_client.py** (on KALI). The client should create a file (any text file with some content will suffice) and send the data in this file to the server. The server should receive the data and write it to a file. The resulting file should be exactly the same as the file on the client side. Once the transfer is complete, the connection should be closed. The output format and behavior should match the example below.



Contents of example.txt



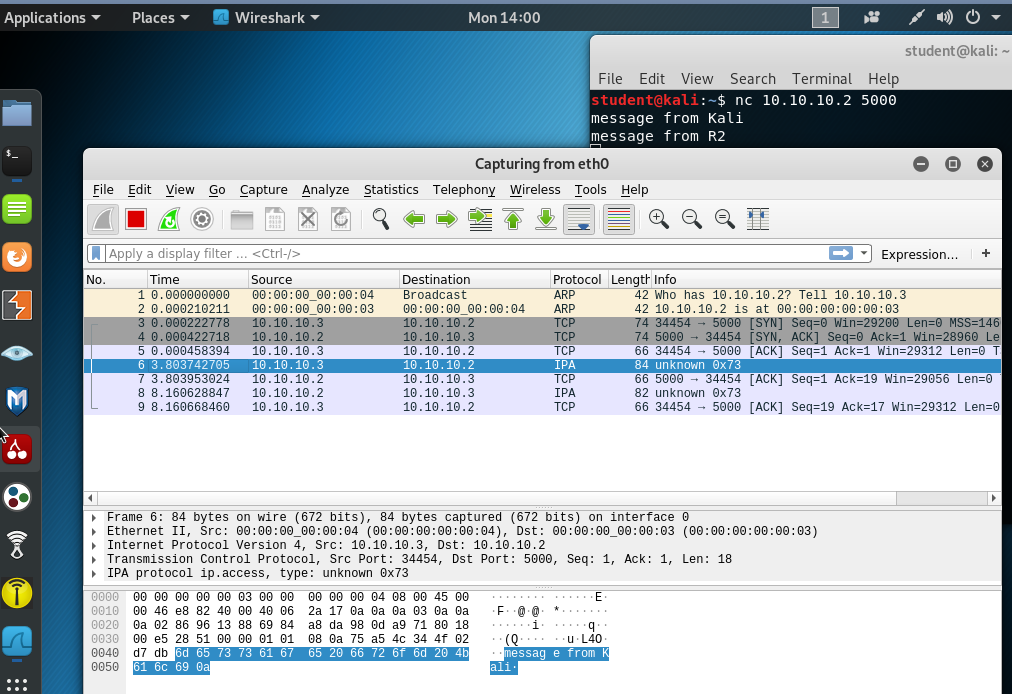
Output is written into the text file serverOutput.txt

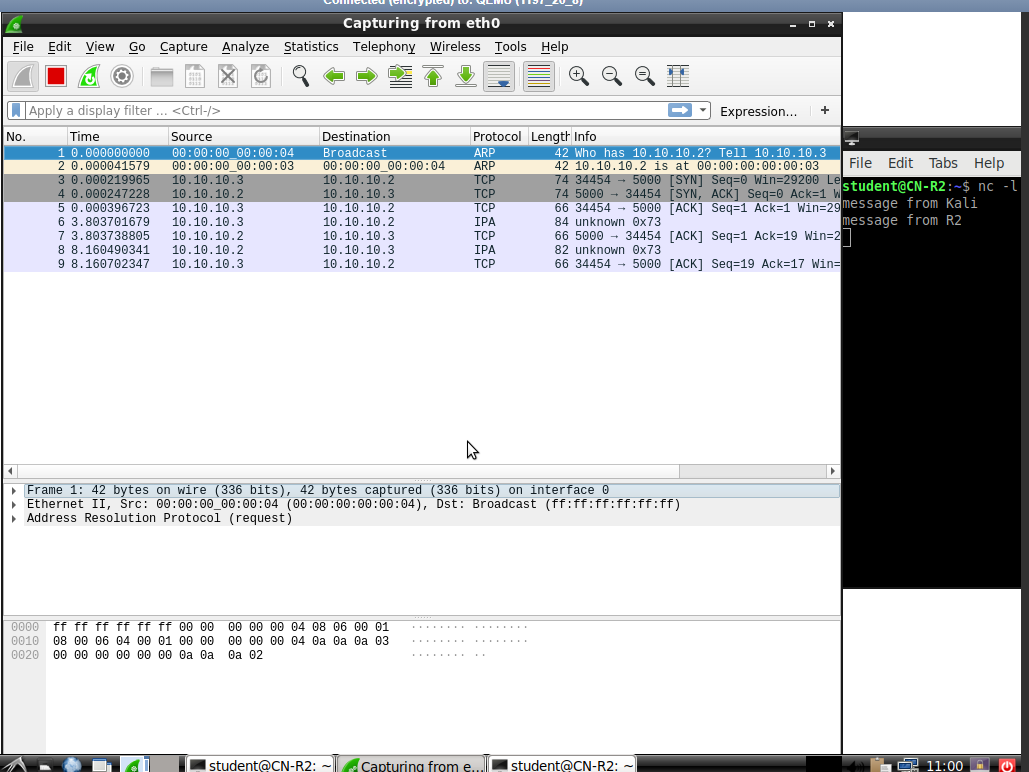


**Part 4: Questions**

1. In netcat, you specified the port on which the server should listen but did not specify the port the server should use to send a message to the client. Which client port does your netcat server send to? Use Wireshark to answer the question and include a screenshot.

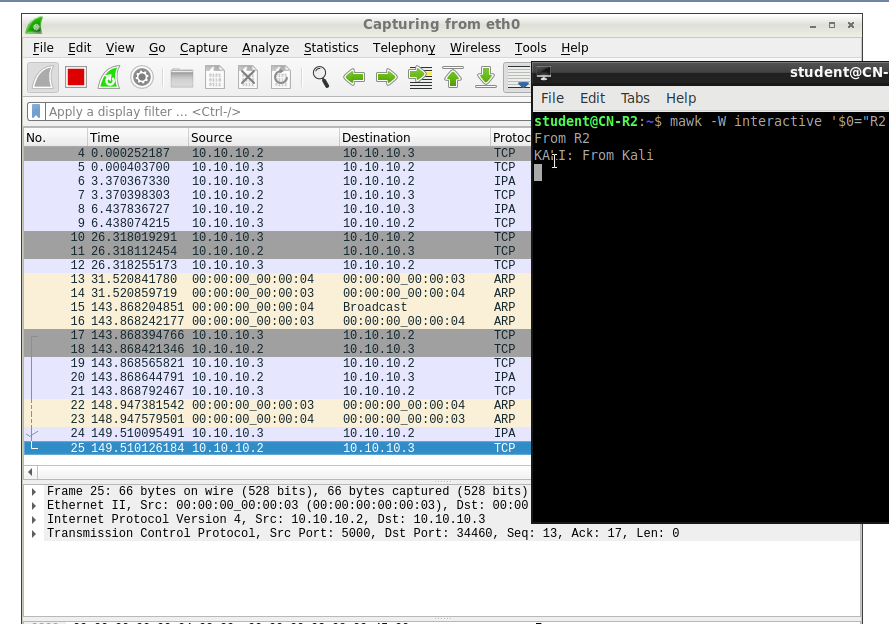
The client has a port of 34454.

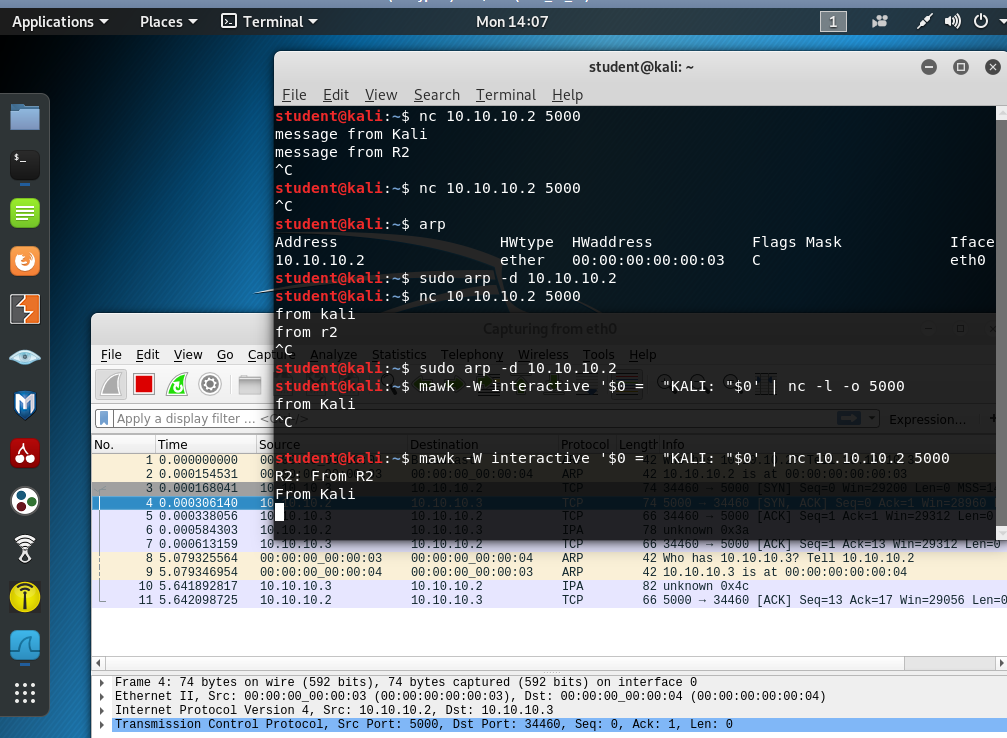




When using mawk -W along with netchat:

The port number becomes 34460. This is because the operating system will automatically assign the client a randomly available port number to use before establishing a connection with the server.





1. Briefly explain your code from Part 2 and Part 3. In your explanation, focus not on the syntax but on the TCP communication establishment and flow.

The server creates a socket and with the specifications that it will run IPv4 and a TCP socket stream. The server’s socket will also do a system call to bind to a specified, open port number then wait for any incoming requests to that port number.

The client will initiate its own socket with specifications to use IPv4 and TCP socket streams through AF\_INET and SOCK\_STREAM. The client connects to the specified server IP address and port number using connect(). The server will see the incoming request, and the client and server will initiate a 3-way handshake[[1]](#footnote-1).

In the 3-way handshake, the client will send a request to initiate a connection to the server. The server will receive the request and send an acknowledgement packet to the client, informing the client that the server is able to establish a connection. The client will receive the acknowledgement response from the server and send its own acknowledgement response to the server. When the server receives the acknowledgement from the client, then the connection between the host and client has been created[[2]](#footnote-2).

TCP handshake is completed after the server’s socket’s accept() call. The server will create a new socket dedicated towards responding to any requests from this specific client machine. The server and client are now ready to communicate to each other.

**Part 2:**

The client will encode its echo message into a byte-sized buffer before sending it to the server. The client will create a TCP packet and transmit the buffer through the client’s socket. The server receives the message and decodes the message back into a string. The server will check the string to see if the word SECRET is found.

If SECRET is found, then it will check and record any the digits as well as the total amount of digits found in the string and then encode a message into a byte-sized buffer. The server will send this buffer in a TCP packet to the client.

If SECRET was not found, then the server returns a TCP packet that informs the client that the secret code was not found.

The client will receive the message from the server and print out the response on its own machine before closing its own socket.

The server closes its socket with the client, ending the TCP connection.

**Part 3**:

The client will read from the text file specified by the user and send the contents of the file through TCP packets. The server will receive the file contents received from the TCP packets. The server will check the buffer received from the TC packet to see if the contents can be written to file.

If the contents cannot be written to the file, then the server will just close the connection. If the contents can be written to file, then the server will open/create an empty text file with the text file name specified by the code. The contents from the TCP packet will be copied into the server’s text file. Once completed, the server will close the file, close the socket with the client, close the server’s TCP socket, and then exit the python file. The client will, in turn, close its own connection socket as well.

1. What does the socket system call return?

If the socket encounters an error or fails in execution, it returns an error number to show the error and a value of -1. Otherwise, it will return the value of the file descriptor as a non-negative integer[[3]](#footnote-3). The file descriptor is the lowest-numbered file descriptor not currently open for the process[[4]](#footnote-4).

1. What does the bind system call return? Who calls bind (client/server)?

The server calls bind.

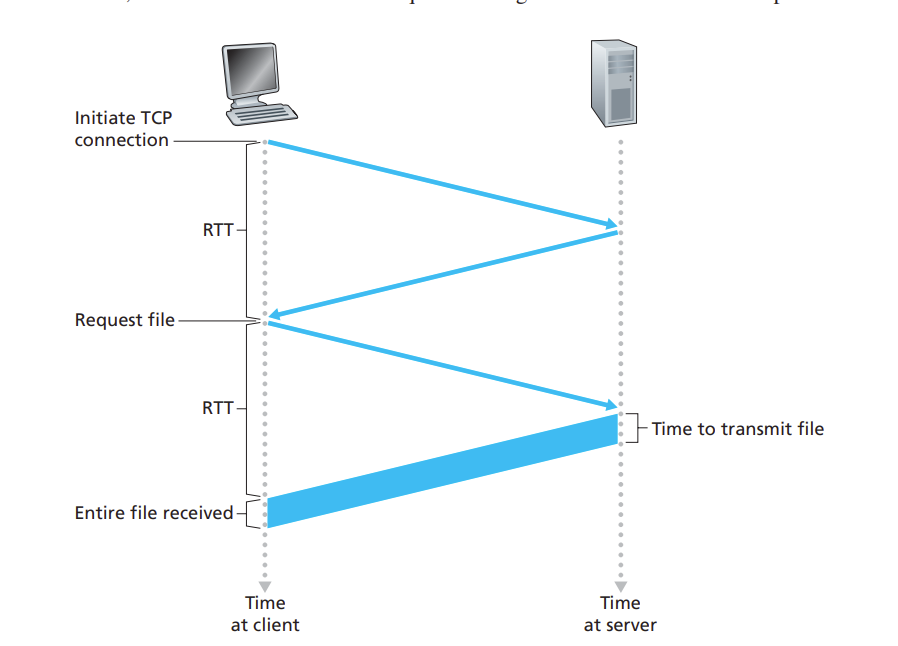
TCP bind will return -1 if an error is encountered and sets the error number. If the bind request was successful, it returns 0[[5]](#footnote-5).

The server has to call an explicit bind request in order to assign an address to the socket and refer it to the file descriptor[[6]](#footnote-6). The client, however, does not need to call bind because the kernel will automatically initiate an implicit bind request when the client connects to the server[[7]](#footnote-7).

1. ) Suppose you wanted to send an urgent message from a remote client to a server as fast as possible. Would you use UDP or TCP? Why? (Hint: compare RTTs.)

TCP has a higher RTT time due to the mandatory three-way handshake process needed. TCP requires a minimum of 2 RTT because it is connection oriented. The first RTT is overhead needed to set up the connection while the second RTT is needed to transmit the message itself.

UDP is a connectionless service, so it does not bother with any handshaking protocol. It will send as much data at any rate, meaning that UDP will transmit the message in only one RTT[[8]](#footnote-8).



1. What is Nagle’s algorithm? What problem does it aim to solve and how?

Nagle’s algorithm is a method to increase the efficiency of a TCP stream by attempting to transmit as many full-sized TCP packets as possible.

When there are multiple small packets under TCP’s maximum byte limit, the server will send out one small packet first. While waiting for the packet to be acknowledged, the server will accumulate the smaller packets and buffer them into one larger packet. When the server receives the acknowledgement from the first packet or if enough packets have accumulated to completely fill the maximum size for a TCP segment, then it will transmit the TCP segment containing the multiple smaller packets. This makes it so that only one packet will be outstanding and make sure that as much data is being sent out in as few RTTs as possible[[9]](#footnote-9)[[10]](#footnote-10)[[11]](#footnote-11).

1. Explain one potential scenario in which delayed ACK could be problematic.

Delayed ACKs conflicts with Nagle’s algorithm extremely poorly. As Nagle mentioned, the 200 ms Ack delay was a solution used to reduce overhead. While every other protocol uses computed time as a guideline, Delayed ACKs is the only solution that uses a fixed time measurement[[12]](#footnote-12).

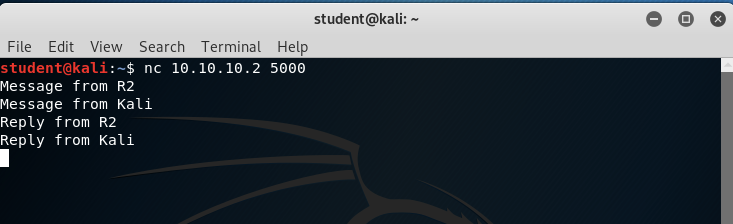
Nagle’s algorithm depends on receiving an ACK to send data while Delayed ACKs want to send more full packets if it can. Combining both will lead to stalling as the client will delay its acknowledgement while the server will delay its packet transmission[[13]](#footnote-13).

If a client using delayed ACKs sends a few packets to a server using Nagle’s algorithm, then it can risk leading to a deadlock between the two. If the client and/or server has issues transmitting packets or transmits multiple packets before reading for any incoming packets, then it could lead to a scenario where the server is waiting on an acknowledgement from the client before sending out more data. The client, however, may not be able to respond because it is too busy waiting to receive any data to piggyback an acknowledgement on.

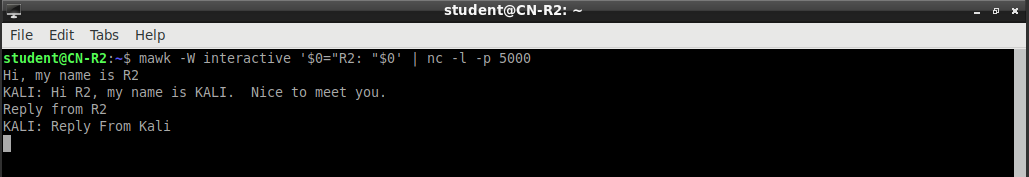
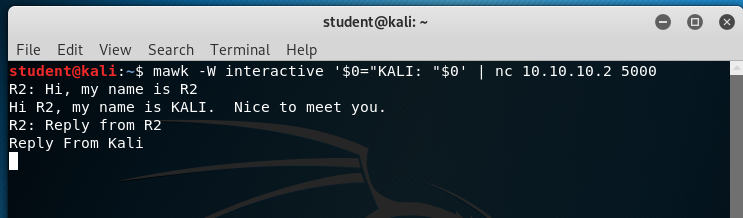
The client is unable to give the signal to the server to send more data, leading to both the client and server waiting for a response from each other[[14]](#footnote-14). This leads to a deadlock between the client and server. If there is a time out feature, then both systems will be bottlenecked and would be transmitting at the maximum amount of time allotted by the timeout system[[15]](#footnote-15).

**Submissions**

1. Screenshots of R2 and KALI showing the netcat TCP chat

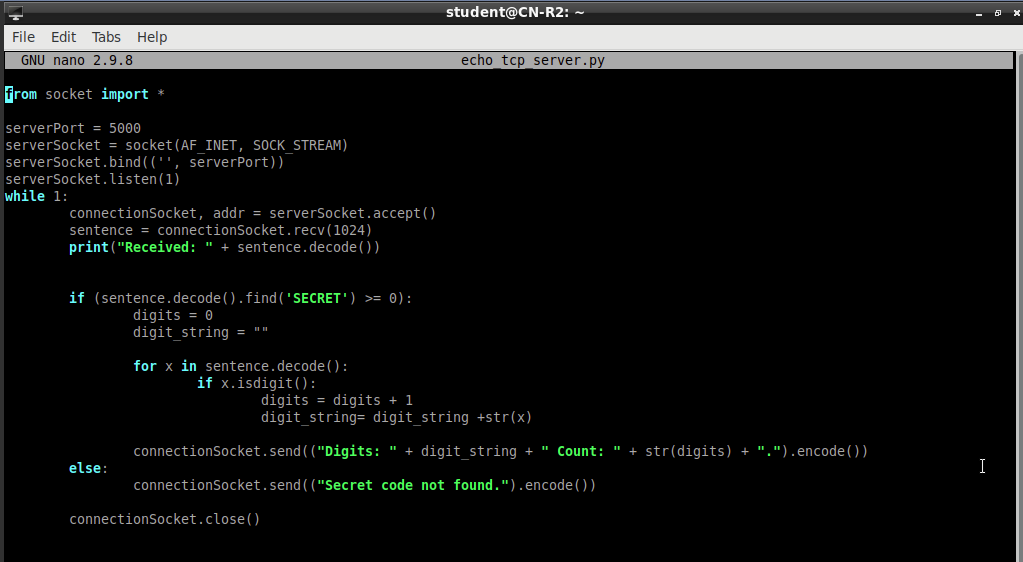


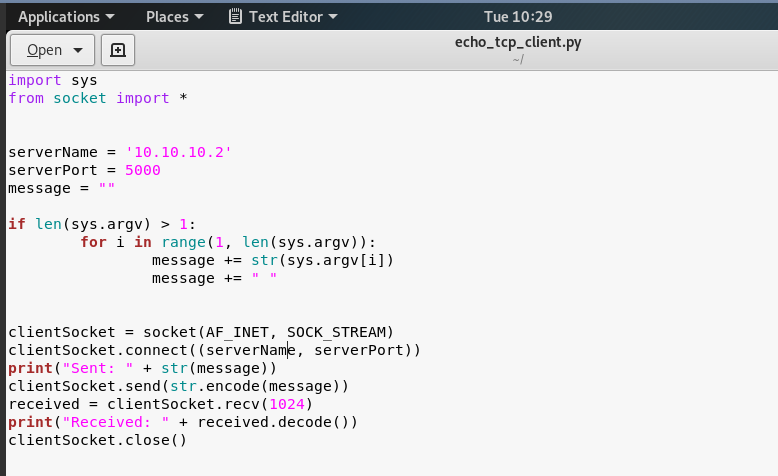
With usernames



2. Screenshots of echo\_tcp\_server.py and echo\_tcp\_client.py

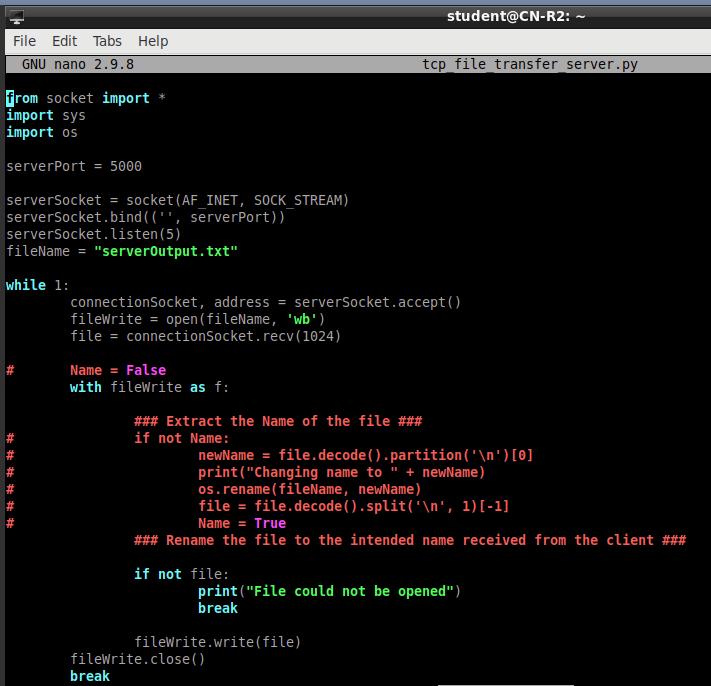
(showing all code)

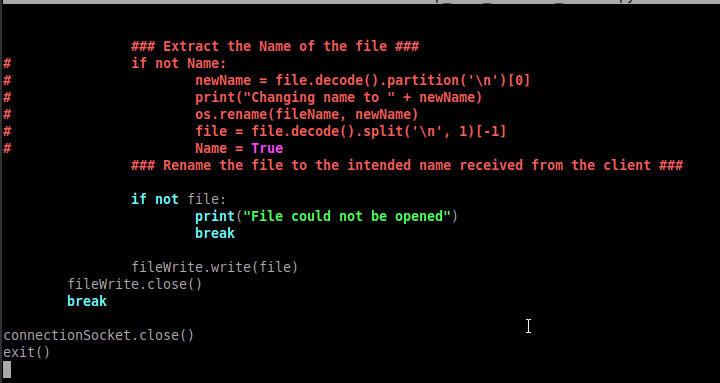


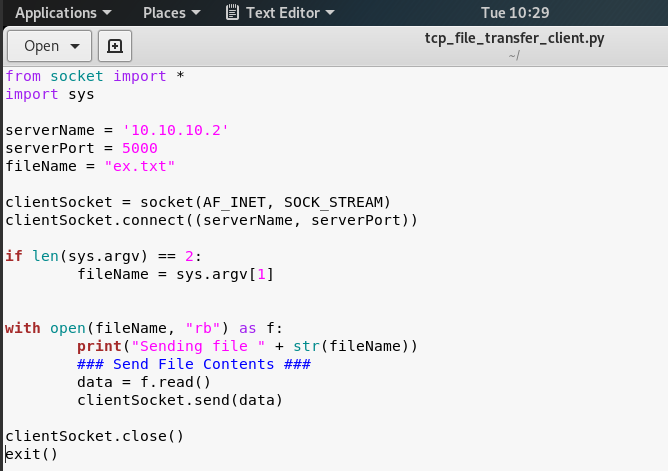


3. Screenshots of tcp\_file\_transfer\_server.py and tcp\_file\_transfer\_client.py

(showing all code)

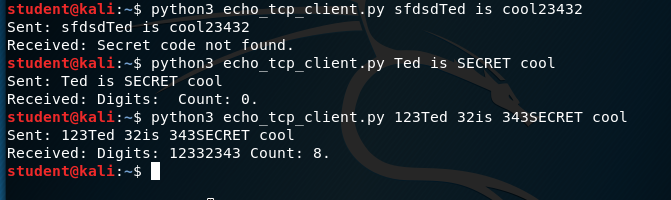






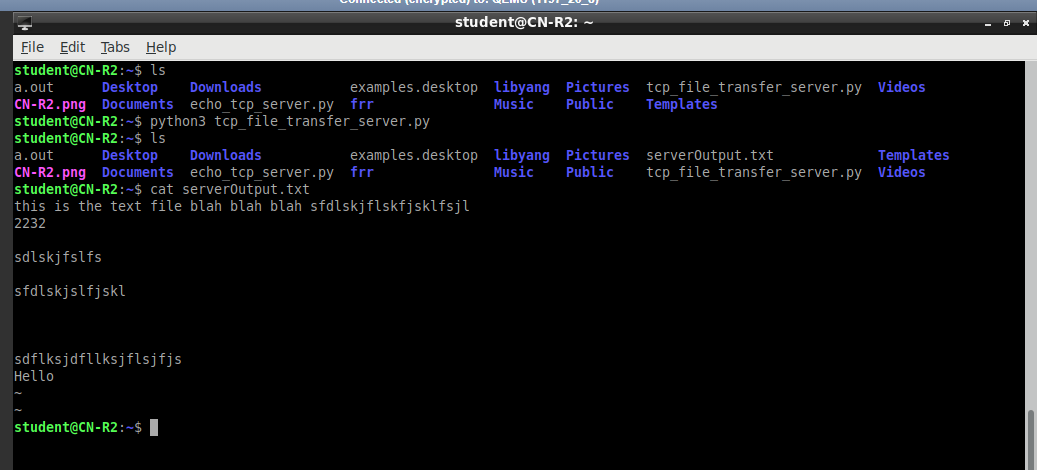
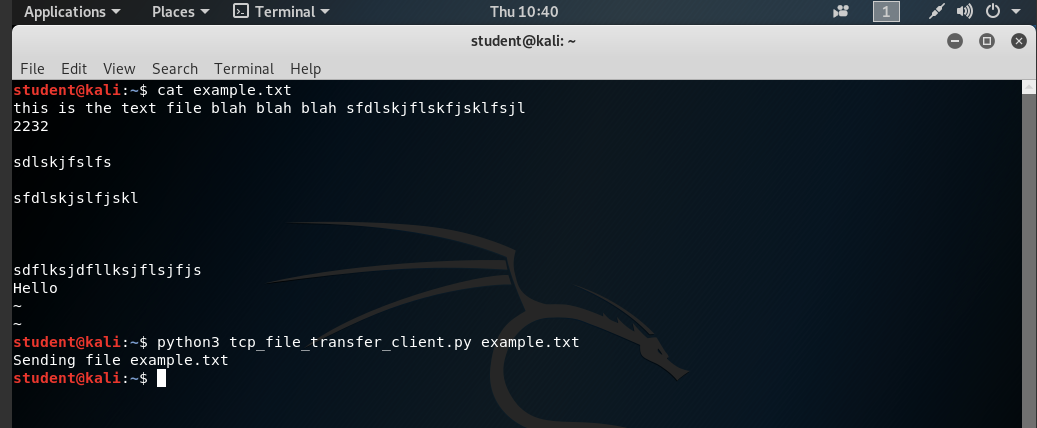
4. Screenshots showing the behavior of Part 2. Make sure to include cases with and

without the secret code.



5. Screenshots showing the file transfer in Part 3: show the original file on KALI, the KALI

terminal after transferring, and the transferred file on R2.



6. Answers to questions 4a-4g

1. Kurose & Ross, pg. 165 [↑](#footnote-ref-1)
2. https://www.techopedia.com/definition/10339/three-way-handshake [↑](#footnote-ref-2)
3. https://pubs.opengroup.org/onlinepubs/009604599/functions/socket.html#:~:text=RETURN%20VALUE,set%20to%20indicate%20the%20error. [↑](#footnote-ref-3)
4. https://man7.org/linux/man-pages/man2/socket.2.html [↑](#footnote-ref-4)
5. https://man7.org/linux/man-pages/man2/bind.2.html#RETURN\_VALUE [↑](#footnote-ref-5)
6. https://linux.die.net/man/2/bind [↑](#footnote-ref-6)
7. https://developer.ibm.com/technologies/systems/articles/au-tcpsystemcalls/#bind [↑](#footnote-ref-7)
8. Kurose and Ross, pg. 102 [↑](#footnote-ref-8)
9. Tanenbaum, pg. 566 [↑](#footnote-ref-9)
10. https://news.ycombinator.com/item?id=9048947 (John Nagle) [↑](#footnote-ref-10)
11. https://www.lifewire.com/nagle-algorithm-for-tcp-network-communication-817932 [↑](#footnote-ref-11)
12. <https://news.ycombinator.com/item?id=9045125> (John Nagle) [↑](#footnote-ref-12)
13. https://www.extrahop.com/company/blog/2016/tcp-nodelay-nagle-quickack-best-practices/#1 [↑](#footnote-ref-13)
14. https://serverfault.com/questions/834326/questions-about-nagle-vs-delayed-ack [↑](#footnote-ref-14)
15. Tanenbaum pg. 567 [↑](#footnote-ref-15)